SAPIENZA
UNIVERSITÀ DI ROMA

SAPIENZA UNIVERSITY OF ROME

MASTER'S DEGREE IN CYBERSECURITY

DEPARTMENT OF COMPUTER SCIENCE

Project 3

Fuzz testing of an image manipulation software using *AFL*

Author:
Marco Ruvolo

Matricola number:
1883257

Professor:
Daniele Friolo

Course:
Security in Software Applications

December 30, 2022

Contents

List of Figures

List of Tables

1 Introduction to testing and fuzzing

1.1 Testing

Software testing is a fundamental part of any *SDLC* (Software Development Life Cycle) and it consists in examining the artifacts and the behavior of the software (under test) by validation and verification.

There exist different levels and techniques of software testing:

- SCENARIO TESTING - to ensure that requirements are complete and consistent (e.g., analysing abuse and misuse cases);
- SPECIFICATION TESTING - to prove the specfication is complete and correct (e.g., through a formal proof or symbolic execution);
- IN-LINE TESTING - to verify invariants, rules and that the internal state of the software is allowed by the security policies, expected and self-consistent (e.g., using assertions);
- UNIT TESTING - to locally test software units, modules and components;
- HUMAN INTERACTION TESTING: to ensure the *UI* (User Interface) and error messages are properly usable (e.g., analysing *UI* abuse cases);
- INTEGRATION TESTING - to test logically integrated software units, modules and components as a combined entity;
- FINAL TESTING - to run test cases against the entire project (e.g., injecting random input);
- ACCEPTANCE TESTING - to determine whether the requirements of the specification and customer contract are met.

In general, testing does one of the following:

- it looks at correct or wanted behaviour for sensible input or some input on borderline conditions (i.e., normal testing);
- it looks for wrong or unwanted behaviour for *bizarre* input (i.e., security testing).

Since a normal use of the system is more likely to reveal functional problems than security problems, users complain about the former ones while malicious users never complain about the latter ones.

1.2 Fuzz testing

Fuzzing, also known as fuzz testing, was proposed by Barton Miller[1] at the University of Wisconsin in 1988. Basically, fuzz testing consists of running the software application to test with random input, identifying the eventual crashes and correlating the input that caused the crashes: it is an effective technique for automated security testing. The basic idea is to see whether the application tested crashes with semi-automatically generate random input.

Initially, fuzzing generated very long input (to make it likely that buffer overruns cross segmentation boundaries) to see whether the system crashes with segmentation fault. Other typical input that can be used for fuzzing could be:

- long strings;
- empty string;
- maximum and minimum integer values;
- zero and negative integer values;
- null values;
- newline, *EOF* (End Of File) and format string characters;
- semicolons, slashes, backslashes, single quotes, double quotes;
- application specific keywords (e.g., *DROP TABLE* for *SQL*).

Nowadays, there have been developed several kind of fuzzing tools which use *smarter* techniques to generate input:

- (blackbox) mutation-based fuzzers, in which random mutations are applied to existing valid input;
- (blackbox) generation-based (also known as grammar-based fuzzers), in which input is generated from some knowledge of format or protocol (e.g., grammar defining legal input, data format specifications);
- (greybox) evolutionary fuzzers, that perform mutations using a genetic algorithm;
- whitebox fuzzers, that analyze the code to determine interesting input (e.g., using symbolic execution).

Tha main weaknesses of generation-based and mutation-based fuzzers are that a huge amount of work is needed to set up the fuzzers and the chance that random changes in input hit interesting cases is small, respectively.

Thus, an evolutionary approach overcomes these downsides: the basic idea is to learn interesting mutations based on measuring code coverage (i.e., if a mutation of the input triggers a new execution path through the code, then it is an interesting mutation and it is kept else it is discarded).

One of these evolutionary tools will be described in the following subsection.

1.3 A fuzz testing tool: *AFL*

AFL[2] (American Fuzzy Lop) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.

In particular, the mutation strategies used include the following[3].

- *bitflip L/S* - deterministic bit flips: *L* bits are toggled at any given time, walking the input file with *S*-bit increment (the current *L/S* variants are: 1/1, 2/1, 4/1, 8/8, 16/8, 32/8).
- *arith L/8* - deterministic arithmetics: the fuzzer tries to subtract or add small integers to 8-, 16-, and 32-bit values (the stepover is always 8 bits).
- *interest L/8* - deterministic value overwrite: the fuzzer has a list of known *interesting* 8-, 16-, and 32-bit values to try (the stepover is 8 bits).
- *extras* - deterministic injection of dictionary terms: this can be shown as ”user” or ”auto”, depending on whether the fuzzer is using a user-supplied dictionary or an auto-created one.
- *havoc* - a sort-of-fixed-length cycle with stacked random tweaks: the operations attempted during this stage include bit flips, overwrites with random and *interesting* integers, block deletion, block duplication, plus assorted dictionary-related operations (if a dictionary is supplied in the first place).
- *splice* - a last-resort strategy that kicks in after the first full queue cycle with no new paths: it is equivalent to *havoc*, except that it first splices together two random inputs from the queue at some arbitrarily selected midpoint.

Please check the technical ”whitepaper” for *afl-fuzz*[4], to explore in-depth technical details and benchmarks.

1.4 Project repository

The project track carried out, the results collected, the plots within this report and this latter itself, are collected in an online *GitHub*[5] repository.

For further information please visit the following link: https://github.com/mrcruv/afl_fuzzing.

2 A case study: *ImageMagick*

ImageMagick[6] is a free and open-source software suite for displaying, converting, and editing raster image and vector image files that can read and write over two hundred image file formats, and can support a wide range of image manipulation operations.

ImageMagick is written in *C*, it is available for a wide range of operating systems, including *Linux*, *macOS*, and *Windows* and it can be used as a standalone application, or as a library that can be integrated into other software programs.

The testing phases of the *ImageMagick-6.5.4.-10*[7] (legacy) release, using *AFL*, are reported in the following subsections.

2.1 Program instrumentation for use with *AFL*

Since the source code of the major *ImageMagick* releases are available on the *ImageMagick*'s official site (for further information please visit the following link: https://imagemagick.org/archive/releases/), the target program is compiled as follows[8][9].

```
$ CC=<path_to_afl_gcc> CXX=<path_to_afl_g++> ./configure --disable-shared
$ AFL_HARDEN=1 AFL_INST_RATIO=100 make clean all
$ make install
```

To link the executable that reads data from a file and passes it to the *ImageMagick* library against this latter itself (or to make sure that the correct .so file is loaded at runtime), a static build is performed (i.e., *–disable-shared* flag). Moreover, *AFL_HARDEN* is set to 1 (i.e., *AFL_HARDEN=1*) in order to enable code hardening options that make it easier to detect simple memory bugs[9][10] and *AFL_INST_RATIO*[10] is set to 100 (i.e., *AFL_INST_RATIO=100*) for allowing the generation of instrumentation for 100% of branches.

2.2 Initial test cases choice

The fuzzer requires an initial test cases corpus that contains a representative sample of the input data normally expected by the targeted application in order to operate correctly. The two basic rules to follow are the following[9]:

1. keep the files small, for a matter of performance (under 1 kB is ideal, although not strictly necessary);
2. use multiple tes cases only if they are functionally different from each other.

The *afl-cmin* utility is used to identify a subset of functionally distinct test cases that exercise different code paths in the target binaries, as follows.

```
$ afl-cmin -i <testcase_dir> -o <new_testcase_dir> -- <tested_program> [...]
```

Indeed, a subset of *PNG* images is drawn out from the *png* directory inside the *afl_testcases* directory[11], to arrange an initial test cases corpus. More precisely, the *afl-cmin* utility is used on the images located on the *afl_testcases/png/full/images* path.

Please note that the *afl-tmin* utility may be used to minimize each test case in a test cases corpus, as follows.

```
$ afl-tmin -i <testcase> -o <new_testcase> -- <tested_program> [...]
```

2.3 Fuzzing binaries

The fuzzing process itself is carried out by the *afl-fuzz* utility, which requires a read-only directory with initial test cases, a separate directory to store its findings and a path to the binary to test.

For target binaries that accept input directly from *stdin*, the usual syntax is as follows[8][9].

```
$ afl-fuzz -i <testcase_dir> -o <findings_dir> -- <tested_program> [...]
```

For programs that take input from a file, marking the location in the target's command line where the input file name should be placed is needed, using "@@", as follows[8][9].

```
$  afl-fuzz -i <testcase_dir> -o <findings_dir> -- <tested_program>@@ [...]
```

Please note that the flags *-t* and *-m* may be used to override the default timeout and memory limit for the executed process, if needed.

Since the *C* function tested in this case study takes input from files, the latter syntax is used.

2.4 Parallelize fuzzing

In order to fully exploit the hardware (since a multi-core system is used), fuzzing parallelization is necessary: every instance of *afl-fuzz* takes up roughly one core[9].

For parallelizing a single job across multiple cores on a local system, *N* instances of *afl-fuzz* should be ran, as follows[12].

```
$ afl-fuzz -i <testcase_dir> -o <sync_dir> -M fuzzer01 -- <tested_program> [...]
$ afl-fuzz -i <testcase_dir> -o <sync_dir> -S fuzzer02 -- <tested_program> [...]
$ ...
$ afl-fuzz -i <testcase_dir> -o <sync_dir> -S fuzzer0N -- <tested_program> [...]
```

The first fuzzer is the *master* instance, the following ones are *secondary* instances. The output directory is shared by all the instances of *afl-fuzz* and each fuzzer will keep its state in a separate subdirectory. The difference between the *-M* and *-S* modes is that the *master* instance will still perform deterministic checks, while the *secondary* instances will proceed straight to random tweaks. Please note that running multiple *master* instances is usually wasteful, although there is an experimental support for parallelizing the deterministic checks.

Monitoring the progress of the jobs from the command line is possible, using the *afl-whatsup* tool as follows[12].

```
$ afl-whatsup <sync_dir>
```

Furthermore, graphs for any active fuzzing task can be generated using the *afl-plot* utility, as follows[9].

```
$ afl-plot <state_dir> <output_dir>
```

2.5 Output interpretation

The fuzzing process will create and update, in real time, three subdirectories (actually, for each fuzzer instance) within the output directory[9]:

- queue: test cases for every distinctive execution path, plus all the starting files given by the user;
- crashes: *unique* test cases that cause the tested program to receive a fatal signal (e.g., *SIGSEGV*, *SIGILL*, *SIGABRT*), grouped by the received signal itself;
- hangs: *unique* test cases that cause the tested program to time out.

Any existing output directory can be also used to resume aborted jobs, as follows[9].

```
$ afl-fuzz -i- -o <findings_dir> -- <tested_program> [...]
```

Please note that crashes and hangs are considered *unique* if the associated execution paths involve any state transition not seen in previously-recorded faults.

3 Result analysis

Convert is the function tested in the case study analysed in this report: three fuzzer instances are ran (one *master* instance and two *secondary* instances) and the minimised test cases corpus previously described (i.e., the minimised *afl_testcases/png/full/convert_min* corpus) is used.

The results carried out from the fuzzing process are quantitatively collected in table 1 and discussed below.

FUZZER INSTANCE	# FILES USED	# MUTATIONS GENERATED	TIME NEEDED	# CRASHES FOUND
convert_master (M)	6	6438	$\simeq$ 3days	715 (36 *unique*)
convert_slave1 (S)	6	7151	$\simeq$ 3days	6753 (396 *unique*)
convert_slave2 (S)	6	7114	$\simeq$ 3days	6590 (386 *unique*)
TOTAL	6	20703	$\simeq$ 3days	14058 (818 *unique*)

Table 1: overall results

Here follow some graphs representing the execution trend of the three fuzzer instances ran.

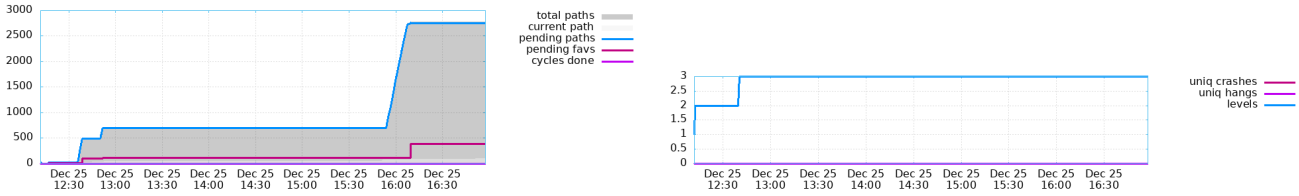

Figure 1: execution trend of the *master* instance *convert_master* for the first 4 hours of processing

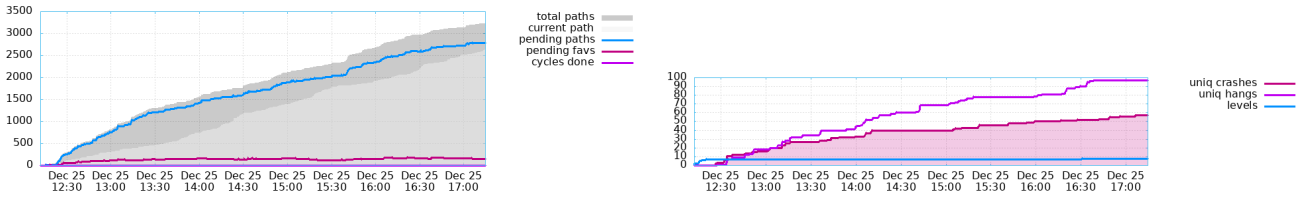

Figure 2: execution trend of the first *secondary* instance *convert_slave1* for the first $4\frac{1}{2}$ hours of processing

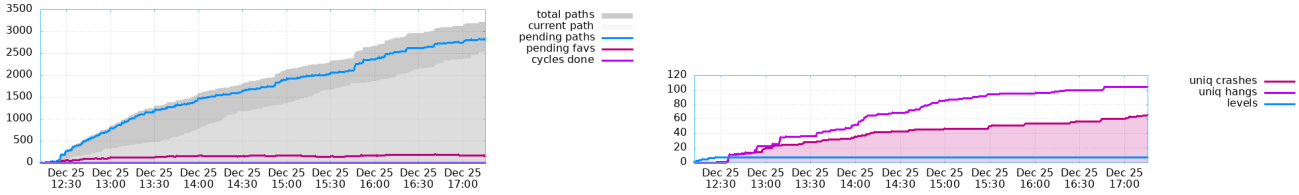

Figure 3: execution trend of the second *secondary* instance *convert_slave2* for the first $4\frac{1}{2}$ hours of processing

3.1 Dynamic result analysis

The crashes identified by the fuzzing process are grouped by the received signal, for instance[13]:

sig·11 : *SIGSEV* (i.e., invalid memory access - segmentation fault);

sig·07 : *SIGBUS* (i.e., bus error);

sig·06 : *SIGABRT* (i.e., abnormal termination condition).

Moreover, the file names for crashes and hangs are correlated with parent, non-faulting queue images[9]: thus, a file named as ”id·000000,sig·06,src·000001,op·splice,rep·128” identifies a crash, having ”000000” as identifier,

correlated with the parent image whose identifier is ”000001”, obtained through the *splice* operation (with 128 bits-rep), that causes a *SIGABRT* signal.

Finally, *Valgrind*[14] (actually, the *Memcheck* memory error detector[15]) is used to better analyse the crashes detected, as follows.

```
$ valgrind <tested_program> [...]
```

Memcheck issues a range of error messages:

- illegal read / illegal write errors[16] → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd);
- use of uninitialised values[17] → Conditional jump or move depends on uninitialised value(s);
- use of uninitialised or unaddressable values in system calls[18] → Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd);
- illegal frees[19] → Invalid free() (Address [...] is [...] bytes inside a block of size [...] free'd);
- when a heap block is freed with an inappropriate deallocation function[20] → Mismatched free() / delete / delete [] (Address [...] is [...] bytes inside a block of size [...] alloc'd);
- overlapping source and destination blocks[21] → Source and destination overlap in memcpy([...]);
- fishy argument values[22] → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] .

For the sake of brevity, just a small subset of the crashes collected by the fuzzer instances is analysed: one file for each $\langle sig, op\rangle$/$\langle sig, op, rep\rangle$ tuple and one file for each $\langle sig, op, rep\rangle$ tuple, w.r.t. the master instance and the secondary instances, respectively. Furthermore, memory leaks[23] are not considered in this report, for the same reason.

3.1.1 *Master* instance: *convert_master*

sig:11,src:000370,op:havoc,rep:2 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig:11,src:000375,op:havoc,rep:4 → Syscall param unlink(pathname) points to unaddressable byte(s) (Address [...] is [...] bytes inside a block of size [...] free'd; Block was alloc'd at [...]); convert: unable to extend cache [...] : File too large @ [...] .

sig:11,src:000375,op:havoc,rep:8 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig:11,src:000791,op:havoc,rep:16 → Killed.

sig:11,src:000791,op:havoc,rep:32 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig:11,src:000462,op:havoc,rep:64 → Killed.

sig:11,src:000474,op:havoc,rep:128 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig:11,src:000613,op:flip1,pos:35 → convert: Unexpected end-of-file [...] @ [...] .

sig:11,src:000368,op:flip4,pos:6 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig:11,src:000368,op:flip16,pos:4 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig:11,src:000613,op:arith8,pos:1,val:+19 → convert: unable to read image data [...] @ [...] ; convert: missing an image filename [...] @ [...] .

sig:11,src:000126,op:ext_AO,pos:2 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

3.1.2 *Secondary* instance: *convert_slave1*

sig·11,src·000151+000416,op·splice,rep·2 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·001868+001627,op·splice,rep·4 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000041+000090,op·splice,rep·8 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000041+000111,op·splice,rep·16 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000390+000779,op·splice,rep·32 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000539+001030,op·splice,rep·64 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000539+001030,op·splice,rep·128 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000127,op·havoc,rep·2 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000129,op·havoc,rep·4 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000127,op·havoc,rep·8 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000127,op·havoc,rep·16 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000242,op·havoc,rep·32 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000212,op·havoc,rep·64 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000291,op·havoc,rep·128 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·06,src·002637+003167,op·splice,rep·2 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd)[1]; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002457+002963,op·splice,rep·4 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002637+003167,op·splice,rep·8 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd)[1].

sig·06,src·002361+002347,op·splice,rep·16 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002637+003167,op·splice,rep·32 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd)[1]; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002541+000333,op·splice,rep·64 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·005324+003962,op·splice,rep·128 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·003816,op·havoc,rep·4 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002413,op·havoc,rep·8 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·003132,op·havoc,rep·16 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·005795,op·havoc,rep·32 → Invalid write of size [...] (Address [...] is [...] bytes after a block of size [...] alloc'd)[1]; Use of uninitialised value of size [...] [1]; Invalid read of size [...] (Address [...] is [...] bytes after a block of size [...] alloc'd)[1]; Invalid read of size [...] (Address [...] is [...] bytes before a block of size [...] in arena "client")[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd)[1]; Syscall param unlink(pathname) points to unaddressable byte(s) (Address [...] is [...] bytes inside a block of size [...] free'd; Block was alloc'd at [...]); convert: unable to extend cache [...] : File too large @ [...] ; sh: 1: rawtorle: not found; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002332,op·havoc,rep·64 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd)[1]; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·07,src·006053+005307,op·splice,rep·32 → Invalid write of size [...] (Address [...] is [...] bytes after a block of size [...] in arena "client"); valgrind: [...] : Assertion [...] failed; valgrind: Heap block lo/hi size mismatch: [...] .

3.1.3 *Secondary* instance: *convert_slave2*

sig·11,src·000182+000386,op·splice,rep·2 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000155+000386,op·splice,rep·4 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000441+001037,op·splice,rep·8 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000319+000435,op·splice,rep·16 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000345+000602,op·splice,rep·32 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000447+000886,op·splice,rep·64 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000246+000582,op·splice,rep·128 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000139,op·havoc,rep·2 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000892,op·havoc,rep·4 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000139,op·havoc,rep·8 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000139,op·havoc,rep·16 → convert: unable to extend cache [...] : File too large @ [...] ; Syscall param unlink(pathname) points to unaddressable byte(s) (Address [...] is [...] bytes inside a block of size [...] free'd; Block was alloc'd at [...]).

sig·11,src·000043,op·havoc,rep·32 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000536,op·havoc,rep·64 → Argument 'size' of function malloc has a fishy (possibly negative) value: [...] ; Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·11,src·000926,op·havoc,rep·128 → Invalid read of size [...] (Address [...] is not stack'd, malloc'd or (recently) free'd).

sig·06,src·002519+002803,op·splice,rep·2 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,srcv001824+002571,op·splice,rep·4 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd)[1]; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·003597+003572,op·splice,rep·8 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd); Use of uninitialised value of size [...] ; convert: Invalid colormap index [...] @ [...] .

sig·06,src·002373+000953,op·splice,rep·16 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd); Use of uninitialised value of size [...] ; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002368+001235,op·splice,rep·32 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·005745+003486,op·splice,rep·64 → Invalid write of size [...] (Address [...] is [...] bytes after a block of size [...] alloc'd)[1]; VALGRIND INTERNAL ERROR: Valgrind received a signal 11 (SIGSEGV) - exiting [...] ; valgrind: the 'impossible' happened: Killed by fatal signal.

sig·06,src·002452+001646,op·splice,rep·128 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·004130,op·havoc,rep·2 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002536,op·havoc,rep·4 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002377,op·havoc,rep·8 → Invalid read of size [...] (Address [...] is [...] bytes after a block of size [...] alloc'd)[1]; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·003137,op·havoc,rep·16 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd); Use of uninitialised value of size [...] ; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·003137,op·havoc,rep·32 → convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

sig·06,src·002373,op·havoc,rep·64 → Conditional jump or move depends on uninitialised value(s)[1]; Syscall param write(buf) points to uninitialised byte(s) (Address [...] is [...] bytes inside a block of size [...] alloc'd); Use of uninitialised value of size [...] ; convert: Unexpected end-of-file [...] : No such file or directory @ [...] .

3.2 Bug fixing

Since the *ImageMagick* release tested has been hugely fixed, the crash-leading files previously detected do not make the most recent version of (legacy) *ImageMagick* (i.e., *ImageMagick-6.9.12-71*) crash: the flaws found are clearly known (removed) vulnerabilities[24][25][26][27][28][29][30][31][32].

[1] #error messages > 1.

4 References

[1] *Prof. Barton P Miller's Home Page*. URL: https://pages.cs.wisc.edu/~bart/.

[2] *american fuzzy loop*. URL: https://lcamtuf.coredump.cx/afl/.

[3] *https://lcamtuf.coredump.cx/afl/status_screen.txt*. URL: https://lcamtuf.coredump.cx/afl/status_screen.txt.

[4] *https://lcamtuf.coredump.cx/afl/technical_details.txt*. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt.

[5] *GitHub*. URL: https://github.com/.

[6] *ImageMagick – Convert, Edit, or Compose Digital Images*. URL: https://imagemagick.org/.

[7] *ImageMagick (legacy) – Convert, Edit, or Compose Digital Images*. URL: https://legacy.imagemagick.org/.

[8] *https://lcamtuf.coredump.cx/afl/QuickStartGuide.txt*. URL: https://lcamtuf.coredump.cx/afl/QuickStartGuide.txt.

[9] *https://lcamtuf.coredump.cx/afl/README.txt*. URL: https://lcamtuf.coredump.cx/afl/README.txt.

[10] *AFL/env_variables.txt at master · google/AFL*. URL: https://github.com/google/AFL/blob/master/docs/env_variables.txt.

[11] *https://lcamtuf.coredump.cx/afl/demo/*. URL: https://lcamtuf.coredump.cx/afl/demo/.

[12] *AFL/parallel_fuzzing.txt at master · google/AFL*. URL: https://github.com/google/AFL/blob/master/docs/parallel_fuzzing.txt.

[13] *signal(7) — Linux manual page*. URL: https://man7.org/linux/man-pages/man7/signal.7.html.

[14] *Valgrind Home*. URL: https://valgrind.org/.

[15] *Valgrind*. URL: https://valgrind.org/docs/manual/mc-manual.html.

[16] *Valgrind (4.2.1. Illegal read / Illegal write errors)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.badrw.

[17] *Valgrind (4.2.2. Use of uninitialised values)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.uninitvals.

[18] *Valgrind (4.2.3. Use of uninitialised or unaddressable values in system calls)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.bad-syscalls-args.

[19] *Valgrind (4.2.4. Illegal frees)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.badfrees.

[20] *Valgrind (4.2.5. When a heap block is freed with an inappropriate deallocation function)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.rudefn.

[21] *Valgrind (4.2.6. Overlapping source and destination blocks)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.overlap.

[22] *Valgrind (4.2.7. Fishy argument values)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.fishyvalue.

[23] *Valgrind (4.2.8. Memory leak detection)*. URL: https://valgrind.org/docs/manual/mc-manual.html#mc-manual.leaks.

[24] *SEGV in 64-bit and 32-bit platforms 07c8accc - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26456.

[25] *SIGABRT convert - IM 6.9.0-1 - 36b70be2 - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26682.

[26] *SIGABRT - 6.9.0-1 convert - coders/rle.c:540 - (5c825062) - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26699.

[27] *convert - IM 6.9.0-1 Beta - cb1f4fa5 - SIGABRT - coders/rle.c:582 - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26733.

[28] *lt-convert - IM 6.9.0-1 - SIGABRT - 39b80955 - coders/rle.c:538 - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26742.

[29] *SIGABRT - lt-convert - IM6 SVN - coders/viff.c:620 - dad2f0d6 - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26816.

[30] *SIGABRT - lt-convert - IM6 SVN - coders/sun.c:477 - 68e4a715 - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26838.

[31] *SIGABRT - lt-convert - IM6 SVN - coders/sun.c:477 - 4a1d6a6d - Legacy ImageMagick Discussions Archive*. URL: https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=26857.

[32] *Various invalid memory accesses in ImageMagick (WPG, DDS, DCM) — The Fuzzing Project*. URL: https://blog.fuzzing-project.org/46-Various-invalid-memory-accesses-in-ImageMagick-WPG,-DDS,-DCM.html.